AFRL-RY-WP-TR-2009-1279

AUTOMATICALLY PARALLELIZING LEGACY BINARY CODE FOR MULTICORE ARCHITECTURES

David August

Princeton University

Sal Stolfo and Simha Sethumadhavan

Columbia University

Michael Locasto

George Mason University

AUGUST 2009
Final Report

AIR FORCE RESEARCH LABORATORY
SENSORS DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7320
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report was cleared for public release by the Defense Advanced Research Projects Agency (DARPA) and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RY-WP-TR-2009-1279 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//

ALFRED J. SCARPELLI
Project Engineer
Advanced Sensor Components Branch
Aerospace Components & Subsystems
Technology Division

//Signature//

BRADLEY J. PAUL, Chief
Chief, Advanced Sensor Components Branch
Aerospace Components & Subsystems
Technology Division
Sensors Directorate

//Signature//

TODD A. KASTLE
Chief, Aerospace Components & Subsystems
Technology Division
Sensors Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

The views, opinions, and/or findings contained in this article/presentation are those of the author/presenter and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

*Disseminated copies will show "//Signature//" stamped or typed above the signature blocks.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
August 2009	Final	16 June 2008 – 31 August 2009

4. TITLE AND SUBTITLE
AUTOMATICALLY PARALLELIZING LEGACY BINARY CODE FOR MULTICORE ARCHITECTURES

5a. CONTRACT NUMBER

5b. GRANT NUMBER
FA8650-08-C-7851

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
David August (Princeton University)
Sal Stolfo and Simha Sethumadhavan (Columbia University)
Michael Locasto (George Mason University)

5d. PROJECT NUMBER
ARPR

5e. TASK NUMBER
YD

5f. WORK UNIT NUMBER
ARPRYD0E

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Princeton University
1 Nassau Hall
Princeton, NJ 08544-0001

Columbia University

George Mason University

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Sensors Directorate
Wright-Patterson Air Force Base, OH 45433-7320
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency/
Information Processing Techniques Office
(DARPA/IPTO)
3701 N. Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RYDI

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RY-WP-TR-2009-1279

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
PAO Case Number: DARPA 14813; Clearance Date: 19 January 2010. This report contains color.

14. ABSTRACT
Legacy codes must be adapted to multicore in order to sustain the current rate of progress made in science and commerce. The goal of this seedling was to study the suitability of particular automatic parallelization techniques for parallelizing legacy codes. We found that n-gram (i.e., repeated instruction sequences) parallelism does exist, but it is not easily leveraged. We found parallelism in many codes previously thought to be inherently sequential. We believe that efforts that consider input data can help liberate high-order parallelism from almost all such "sequential" codes.

15. SUBJECT TERMS
multiprocessor, multicore, parallelization, compilation, binary optimization

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Alfred J. Scarpelli
Unclassified	Unclassified	Unclassified	SAR	28	19b. TELEPHONE NUMBER *(Include Area Code)* N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39-18

Table of Contents

Section **Page**

List of Figures

Figure **Page**

Acknowledgements

We would like to thank Bill Harrod at DARPA/IPTO for funding this project and AFRL for contracting it. We also would like to thank Al Scarpelli for his support and guidance. This seedling brought together researchers with varied backgrounds and perspectives, who may have otherwise never collaborated. These interactions have already had a lasting impact on the respective works of each researcher.

1 Executive Summary

The industrial adoption of multicore computing has forced legacy code owners to choose between two undesirable choices: either manually parallelize code at significant cost or simply accept significant performance slowdown (because multicore processors typically tend to be clocked slower than their uniprocessor predecessors). If legacy codes cannot be sped up, multicore adoption can jeopardize decades of low-cost, continuous application performance improvements, a key enabler for substantial progress we have made in science and commerce. The goal of this seedling was to study the suitability of certain automatic parallelization techniques for parallelizing legacy codes.

The main thrust of our investigation was to measure the legacy binaries for evidence of latent parallelism in repeated instruction sequences (n-grams). The intent was to manually parallelize repeating code sequences, n-grams, and then build tools to automatically apply parallelizations to the entire code base. While we find that n-gram (i.e., repeated instruction sequences) parallelism does exist, it is not easily leveraged. Such n-gram analysis can, however, assist compiler-based parallelism.

Specifically, our findings show that:

- **Static binary analysis for repeated instruction sequences has limited utility.** Although we find many repeatable code sequences, our results confirm the limited use of statically identifying parallelizable sequences due to complexities of modern code compilation.
- **Dynamic analysis of repeated instruction sequences offers little speedup.** Although we found ample instruction-level parallelism (ILP) in legacy code by using an ideal machine (limited only by data dependencies and not physical resources), repeated code sequences show little ability to be parallelized on real hardware.
- **Semantic pattern-matching shows progress.** Matching parallelizable code patterns shows promise: we developed a prototype system that maps x86 code segments to a representation that is useable in a compiler phase that transforms code into a speculatively parallel version. This approach shows ample parallelism.
- **Input data patterns can reveal dependencies and direct the extraction of parallelism.** We find that one very promising direction for future work is to consider parallelization of legacy code under specific partial input data. An initial study shows significant gains.

2 Introduction

The rise of commodity multicore computing has forced many organizations to re-evaluate current programming paradigms and software execution models. While multicore machines offer exciting possibilities for speeding up naturally parallelizable tasks, the research community still faces a considerable number of fundamental questions in its exploration of viable alternatives for effective and efficient use of these resources. Furthermore, practitioners in a variety of industry and government organizations find themselves grappling with the practical issues stemming from the need to take advantage of the power provided by now-commodity multicore hardware --- a power for which the extensive legacy computing base seems ill-matched, since it was not designed to run on more than a single CPU or at best, a Symmetric Multiprocessing (SMP) machine with only a few independent processors.

Parallel computing is enjoying a rebirth. At this early stage, however, the nature and form of parallel computing elements, compilers, libraries, and programming methods are far from a foregone conclusion. Resurgent research efforts into parallel computing seem driven by and tightly coupled to the advent of multicore processor packages in commodity desktop and server hardware. This hardware offers a boon of readily available computational resources to support parallel workloads. The hope seems to be that research efforts can create methods that automatically identify and extract task level parallelism suitable for execution on these machines. Two fundamental issues exist:

1) How can we automatically convert a program written to execute on a single core to parallel execution on multiple cores? What kinds of methods do we need to study to develop a program, runtime framework, or set of libraries that can support such automatic translation?

2) How can we enable current and legacy software to efficiently leverage the rapid pace of hardware advance? CPU computing power doubles roughly every year, and packing exponentially more cores on a chip is a stated goal of major hardware vendors. The pace of most software development and certification efforts typically lags such increases in hardware capacity. What is the best way to take advantage of such future hardware advances in the software development lifecycle?

The goals of our specific project were to propose, define, and take measurements of legacy binaries to see whether there is evidence of latent parallelism. Furthermore, we determined to use the measures to help define a set of methods for automatically extracting parallel execution from legacy binaries.

3 Methods, Assumptions, and Procedures

This report summarizes the highlights of our progress and results. Our activities have primarily focused on two efforts: (1) collections and analysis of data (memory allocation & access patterns, instruction sequences) traces and (2) understanding the outline and structure of a procedure for identifying parallelizable patterns in binary code.

To this end, we have built a set of program instrumentation tools (using Pin, www.pintool.org) to collect both static and dynamic instruction n-gram traces for Microsoft Windows and Linux user-level software applications. We have also collected Microsoft Windows load/store profiles for Microsoft Word, Excel, and Powerpoint, along with a video transcoder. We collected user-driven profiles of Microsoft Windows events. We have demonstrated a proof-of-concept speedup of x86 code in the GNU Image Manipulation (GIMP) graphics program for a variety of image transformations (for example, we can speed up one filter from seventy seconds of execution time to ten seconds of execution time). We have built a prototype system for understanding how to map parallelizable execution patterns by translating x86 traces to the Low-Level Virtual Machine (LLVM) language. Finally, we have collected malloc-based profiles of a variety of Linux applications to help us see if understanding the data structures or other high-level objects in memory can inform choices about associating tasks or threads with certain high-level data structures (as opposed to specific virtual memory addresses). We have coupled this latter work with an understanding of related work from theoretical Computer Science on the topic of topological sorting and ordering of resource and data requests.

Since our goal is to estimate the potential for automatic hardware/software parallelization of legacy codes on multicore platforms, we developed a methodology suitable for this task. We focused our data collection and analysis on legacy codes and benchmarked both Microsoft and Linux applications (e.g., Firefox, Microsoft Word, Microsoft Internet Explorer). We first estimated the potential for parallelism, then measured the code sequences for n-gram based parallelism, and finally undertook a dependence-based measurement approach.

4 Results and Discussion

We summarize our main findings here and expand on them with supporting data and figures later in this section. We have observed that n-gram approaches provide a coarse measure of self-similarity but cannot easily be leveraged for parallelization patterns. Instead, we have come to the conclusion that parallelization templates offer a target for the translation of x86 binary instruction stream to LLVM. We also note that a user-centered viewpoint offers a way to focus the search for parallelizable code tasks on "high-pain" areas of execution: sections of code that cause a perceptible delay for a user.

4.1 Specific Findings

We find that:

- **Limited power of static binary analysis.** Our results confirm general expectations that simple static analysis of code sequences in legacy code reveals little to no opportunity for automated parallelization. The complexity of the underlying compilation framework and instruction set provides a vast array of options of rendering binary code for the same source code. We abstracted the static analysis to consider classes of instructions rather than exact instruction sequences and yet little additional parallelism was revealed.
- **Analysis of dynamic instruction sequences also offers little speedup.** We extended our analysis to consider dynamically executed instruction sequences and applied the same analytical methods to detect repetitive instruction sequences. The outcomes were as poor as the original simple static analyses.
- **Semantic pattern-matching shows progress.** An alternative approach was investigated following the methods developed by David August and his group to find what we may call "pragmas" that specify how a pattern of instructions in a code sequence can be rewritten or transformed into a parallelized version. This area is ripe for exploitation and further development. Ample parallelism can be discovered in legacy code using this approach.
- **Input data patterns reveal data dependencies and help organize the extraction of task level parallelism.** One particularly intriguing future research path is to consider parallelization of legacy code under particular input data. It is the input data to a complex application that ultimately drives the execution paths of the application and the potential parallelization of those paths of execution. Some work along these lines was explored late in the seedling research program; much more work needs to be done to fully develop this line of thought.

During the course of the research project, we developed a number of tools and artifacts, including:

- A Pin-based tool for extracting both static and dynamic instruction sequences

- A Pin-based tool that provides an analysis and debugging environment based on data-structure representations
- A host-based auditing sensor that monitors user patterns (such monitoring provides one way of focusing on parts of an application that a user commonly executes, and thus may be interesting targets for parallelization or speedup).
- A Pin-based tool for extracting instruction sequences and translating them to the LLVM representation in order to bridge the gap between Princeton's parallelization pattern-matching compiler and the x86 machine language.

4.2 Potential Parallelism in Legacy Codes

To estimate the potential parallelism in a body of code, we devised a method to statically analyze legacy binary code sequences by counting the number of repetitive n-gram subsequences that may be executed in parallel. An n-gram is a contiguous sequence of n instructions. The static analysis scans the code and counts the number of distinct instruction sub-sequences of size n, varying n. The analysis sought to answer the following questions. Are there instruction subsequences that repeat often enough so that parallel execution would speed up the program significantly? Is it easy to dispatch these repeated subsequences to execute efficiently on multicore architectures to realize this inherent parallelism? Furthermore, how many distinct parallelizable subsequences exist? Our measurements indicate that there are instruction sequences that repeat often enough, but they are not easily profitably parallelized across cores, even though many distinct parallelizable sequences exist.

We measured a set of third-party commodity legacy binaries, including Microsoft Notepad, Microsoft Word, and a video transcoding program (all on Microsoft Windows using Pin). As we can see in the Figure 1, there is ample instruction level parallelism in legacy codes given an "infinitely parallel" machine (i.e., a machine constrained only by dataflow constraints, not physical resources). Although this result is encouraging, the question of how to automatically extricate it on real multicore chips remains a daunting challenge.

In Figure 1, the vertical axis shows the fraction of dynamic instructions that can execute each cycle assuming that execution is constrained only by dataflow constraints. The horizontal axis shows the number of time steps required to execute the program. In considering the execution of Notepad on this "ideal" machine, the chart shows that about 54% of the dynamic instructions can execute in the first time step of the program! Thereafter, 56% execute within 10 time steps, 58% within 100 time steps and all of the program instructions can execute within 100000 time steps. Overall, this works out to a limit of instruction level parallelism of 93 instructions per cycle for Microsoft Notepad. Trends are similar for other programs; the ILP limit for Microsoft Word is about 47, and for transcoding it is 165. The question remains how to extract such ILP from legacy binaries in the absence of a machine with an unbounded amount of hardware resources (i.e., a machine that is restricted by the realities of on-chip cache sharing and inter-core communication).

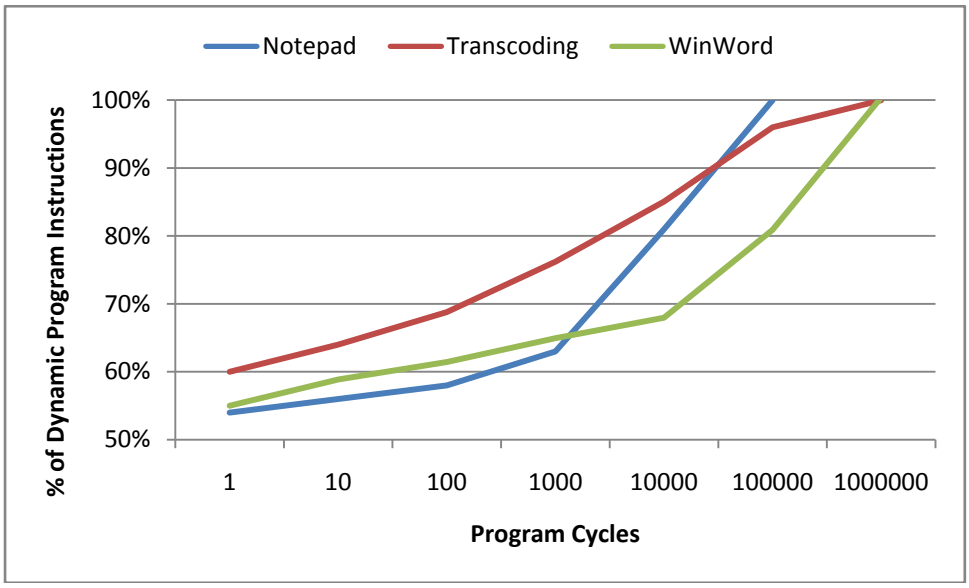

Figure 1. Dynamic Instructions Ready to Execute

4.3 Limited Power of Static and Dynamic Binary Analysis

Our experiments showed that there is some repetition but the length of the repetitive instruction n-grams rarely exceeds more than 11 instructions. Current inter-core communication latencies are not favorable for parallelizing these few instructions across cores. We think, however, that instruction repeatability can be used to synthesize special purpose functional units (accelerators) to speed up these sequences within one core. This is useful because ultimately the time spent in sequential regions will limit the maximum exploitable parallelism (Amdahl's law). In considering our measurements of dynamic n-grams, we see that there isn't a single dominant pattern or larger patterns in these benchmarks, which limits the applicability of n-grams to parallelizing the applications we studied.

These findings imply three results. First, n-gram sequences cannot be effectively parallelized across cores because of inter-core latencies. Second, hardware accelerators that speed up the execution of n-gram sequences *within* cores may be useful. Finally, without the use of such accelerators, program speedup is limited by the time spent inside n-gram instruction sequences themselves.

4.4 Semantic Pattern-matching Shows Progress

Our initial results indicated that using manual emulation of a compiler, scalable performance was found across all C programs of SpecINT2000, a benchmark suite of notoriously sequential programs. By exploiting the pipeline parallelism lurking in sequential applications and using the power of targeted speculation, the Decoupled Software Pipelining (DWSP) (see http://liberty.princeton.edu/Research/DSWP/ for an overview and links to related research papers) approach demonstrated the possibility of

automatically extracting long-running, concurrently executing, streaming threads from unmodified and minimally modified (on the order of 60 lines in 500,000) sequential C programs.

In contrast to DOALL and DOACROSS, which partition the iteration space across threads, DSWP operates by partitioning the loop body into stages of a pipeline, with each stage executing within a thread. These stages are formed in such a way that dependences among them flow in a single direction without any feedback loop. The pipeline stages are executed in parallel on different threads, thus forming a pipeline of threads. Since it keeps dependence recurrences local to a thread, DSWP is highly tolerant of the inter-processor communication latency, as it avoids communication latency on the critical path. DWSP has shown a geometric-mean speedup of 5.5X across 11 SpecINT2000 programs using up to 32 threads, as shown in Figure 2 [BVZJA07].

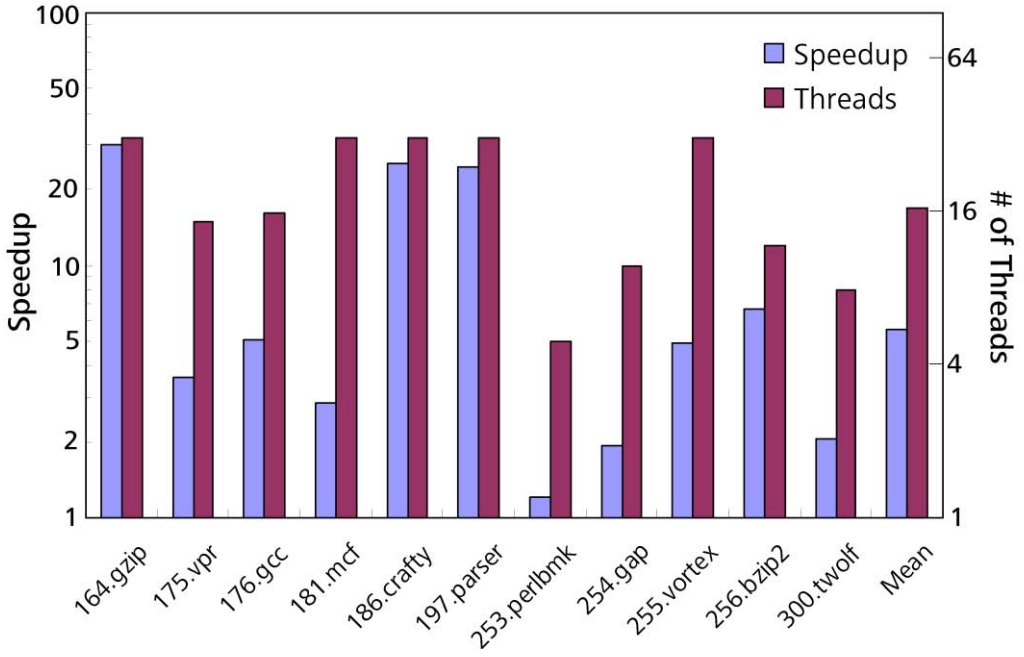

Figure 2. Speedups Obtained With DSWP and Derivative Techniques

We have been able to increase our speedups even further by using DSWP as an enabling transformation for various loop parallelization techniques such as DOALL, LocalWrite and SPECDoall. Unlike DSWP, which tries to maximize performance by balancing the stages of the pipeline, a newly developed optimization DSWP+, tries to assign work to maximize performance by balancing the stages of the pipeline. DSWP+ tries to assign work to stages so that they can be further parallelized by other techniques. After partitioning DSWP+ allocates enough threads to the parallelizable stages to create a balanced pipeline. Our results show that DSWP+ yields more performance than DSWP and other parallelization techniques when applied separately.

4.5 Input-Dependent Parallelism

Prior art has generally focused exclusively on static parallelization of program code. However, we found that some applications cannot be effectively parallelized at compile-time because the input set contains the bulk of the application definition. One example of an application of this type is an interpreter. In these applications, the input set contains the specification of the actual work. In these situations, the scope of parallelization needs to be expanded to include the input program to capture the actual parallelism.

To demonstrate the problem with interpreters more concretely, consider the example Perl program in Figure 3. The Perl code in Figure 3(a) increments all elements of an array a by 1. This is clearly a parallel loop where all iterations can be executed concurrently. However, Figure 3(b) shows a simplified version of the interpreter loop inside of Perl. Each iteration decodes the next Perl operation to execute and performs the actual execution by calling the appropriate processing function. This loop is inherently sequential, as it must keep track of two pointers, the fp pointer that specifies the position in the Perl program and the data pointer that specifies the position in the actual data that is being operated upon. Each of these pointers is updated every iteration to maintain the current position in both the Perl program and input data. As a result, the inherently parallel work in this input is completely sequentialized through the use of a general interpreter.

```
  $i=0
  for(; $i<$n; $i++) {
A:  @a[$i]++;
  }
```
(a) Perl code

```
  while(fp != 0) {
A:  switch (perl_decode(*fp)) {
B:      case 1: fp = perl_array(data);
C:      case 2: fp = perl_inc(data);
D:      case 3: fp = perl_lt(data);
        ...
    }
  }
```
(b) Simplified Perl interpreter loop

```
  while(fp != 0) {
B1:   perl_array(data);
C2:   perl_inc(data);
C3:   perl_inc(data);
D4:   fp = perl_lt(data);
  }
```
(c) Run-time processing loop

Figure 3. An Example to Illustrate the Problems With Parallelizing an Interpreted Program.

To reveal the true parallelism, the Perl program execution, rather than the interpreter, must be analyzed in more depth. At run-time, the sequence of operations (represented as Perl functions) shown in Figure 3(c) is applied to each array element a. For each element, operation B, followed by two instances of operation C, and finally operation D are applied. This sequence is then repeated for each array element until the input is exhausted. Note that each iteration of the loop in Figure 3(c) is independent with the exception of the control dependence from D4 that determines if there are more array elements to process. Thus, there is an opportunity for a run-time system to discover such instruction sequences and parallelize the interpreted application using static methods such as speculative DOALL or DSWP.

To test the viability of this approach, we manually applied dynamic speculative DOALL parallelization to the dynamic dependence graph during execution of the 253.perlbmk benchmark on three reference inputs.[1] Each Perl script is dominated by one loop body. The

[1] The inputs were diffmail.pl, perfect.pl, and splitmail.pl.

result was that performance gains ranging from 1.4-2.6x were observed. This compares favorably to the best static parallelization performance gains of 15%.

In these programs, no amount of profiling can help the compiler determine which set of parallelizations are going to be effective at run time. In fact, the parallelism available may be different for each and every input. To address this problem (or to capture this otherwise lost opportunity), we recommend the development of run-time thread extraction methods specifically for these kinds of programs.

While there are many challenges to overcome to achieve dynamic threading of input-dependent codes at run-time, the basic idea is simple. During execution, a subset of the program dependence graph (PDG) materializes. This dynamic PDG can then be parallelized using the same techniques applied statically. The key is in minimizing the amount of work necessary at run-time. We recommend that this be done in three ways.

First, code that is successfully parallelized statically does not need to be considered. Loops that are found or made to be DOALL, Speculative-DOALL, Parallel-Stage DSWP, or Speculative Parallel-Stage DSWP by the static pass should not be considered since each of these loop types generally provides scalable parallelism in practice. This will relieve the system from considering many, if not most, program loops.

Second, the dynamic hot-path PDG is only a very small subset of the dynamic and static PDGs. Using the 90-10 rule, we might expect only 10% of the code hot enough to be considered important. This hot PDG further consists of two parts: dependences that always exist (or always do not exist) when the code is executed and dependences that are highly input dependent. Dependences of the first type only require hot control path detection that is known to be lightweight. Dependences of the second type should be identified a priori for monitoring.

Finally, run-time transformations need not be as powerful as their static counterparts, and much of their work can be partially evaluated statically. For example, while the static PDG generator must start with the code to generate the PDG, the dynamic PDG generator can start with the static PDG and simply mask out unrealized dependences. Analogs exist in the partitioning, transformation, and code generation process.

Each of these ways should be explored to reduce the cost of run-time threading, in addition to looking for other ways to accomplish this goal.

4.6 Resource Requests Re-orderings

We created a measurement tool that observes a program's execution in a real-time manner and dynamically measures the potentials for safely re-ordering resource requests that the program makes. This goal is the first step toward dynamically parallelizing legacy binary programs. The hypothesis is that the requests to allocate and free resources and data structures are potentially over-constrained. These over-constrained

specifications may be malleable: the machine might be able to split the total resource request sequence into subsets that can be executed in parallel as independent threads. This hypothesis is based on our observation dealing with the commutative function parallelization pattern.

We consider a variant of the classic online topological ordering problem as the abstract model for this application. We have a directed graph G = (V, E), where G is fixed initially. New vertices and edges are added and deleted over time, and we maintain a topological order of all vertices in the graph. Each vertex represents a chunk of allocated memory. The edges among vertices represent the dependency between vertices such that for any edge (i, j) in E, j cannot be executed until i has finished. Given such a graph, any cut (A, B) specifies an arrangement of vertices on the multicores. (A "cut" C is a subset of edges of the graph such that after we remove these edges, the vertices are separated into two parts so that any two vertices in these two parts are not connected.) We can use each vertex to represent each memory allocation operation. An operation that frees memory corresponds to removing one vertex from the graph and its associated edges. Also, as long as one topological ordering is given, any cut on the graph is a safe way to parallelize binary programs.

A topological sort of a directed acyclic graph (DAG) is a linear ordering of all its vertices such that if the graph contains an edge (u, v), then u appears before v in the ordering. If the graph contains a cycle, then no linear ordering is possible. There exist well-known algorithms for computing the topological ordering of a DAG in time O(m + n) in an offline setting [T72, KS74]. These offline algorithms apply depth first search (DFS): call DFS to compute the finishing time for each vertex and then as each vertex is finished, insert it onto the front of a linked list. The final list would be the linear topological order for those vertices.

Algorithms running in quadratic time have been found for online topological ordering [KB06, AFM08, BFG09]. Our goal in this analysis was to update and maintain the topological order over time. We have not found any existing work dealing with this task, especially when we combine topological ordering with maintaining cuts in an online manner.

Topological Ordering Related Work Since topological ordering of resource requests provides a strong underlying theoretical guide to data-centric efforts to automatically extract parallelism, we provide a brief annotated overview of the relevant literature in this space. This literature review should serve as a reference for understanding what limits have been reached by the theoretical Computer Science community in considering the problem of online (i.e., dynamic) topological sorting of unpredictable patterns of resource requests.

In the online variant of the topological sorting problem, the edges of the DAG are not known in advance but are given one at a time. Each time an edge is added to the DAG, we are required to update the topological order. The objective is to minimize the total running time subject to various problem constraints. More specifically, the goal is to

maintain a data structure that supports two operations: (1) edge insertions, in which a new edge is added to the graph; and (2) queries of the form: "Does a vertex *u* come before another vertex *v* in the topological ordering?"

Incremental topological ordering [AFM08] considers the problem of maintaining a topological ordering subject to dynamic changes to the underlying graph. The adversary adds *m* edges to the graph, one edge at a time, and no edges will be deleted. Incremental topological ordering has arisen in a variety of contexts. For example, it is required for incremental evaluation of computational circuits [AHRSZ90], deadlock detection [B90], incremental compilation [MNR93, OLB92] where a dependency graph between modules is maintained to reduce the amount of recompilation performed when an update occurs, and source code analysis [PK06] where the aim is to statically determine the smallest possible target set for all pointers in a program and as a subroutine of an algorithm for incremental evaluation of computational circuits [AHRSZ90]. It is also used as an online cycle detection routine in pointer analysis [PKH03], and the core of constraint systems for stochastic search such as those featured in Comet [MV02].

The naive way of computing an incremental topological order is that in each time slice, we recalculate the new ordering after each edge insertion with the offline algorithm which takes $O(m^2 + mn)$ time in total (where *m* is the number of edges and *n* is the number of vertices.) Marchetti-Spaccamela et al [MNR96] gave the first nontrivial solution, an algorithm that can insert *m* edges in O(mn) time, giving an amortized time bound of O(n) per edge instead of the trivial O(m). This result provides the best amortized time known so far.

The only nontrivial lower bound for online topological ordering is due to Ramalingam and Reps [RR94], who show that an adversary can force any algorithm to perform Ω(n log n) vertex relabeling operations while inserting n-1 edges (and creating a chain). Thus, if the labels are maintained explicitly, this implies an Ω(n log n) bound on runtime. Pearce and Kelly [PK06] gave an algorithm that they showed to be fast in practice on random sparse graphs, but that is provably worse than that Alpern et al. algorithm in the worst case. Kavitha and Mathew [KM07] gave a slightly better variant. Recently, Ajwani, Friedrich and Meyer [AFM08] present a simple algorithm which maintains the topological order under an online edge insertion sequence, in $O(n^{2.75})$ time, independent of the number *m* of edges inserted. For dense DAGs, this is an improvement over the previous best result [KB06]. They also demonstrate empirically that this algorithm clearly outperforms [KB06], [MNR96], [AHRSZ90], and [PK06] on a certain class of hard sequences of edge insertions, while being at most a factor of 2 to 4 away on random edge sequences leading to complete DAGs.

Most recently, Bender, Fineman, and Gilbert [BFG09] present a new algorithm that has a total cost of O(n^2 log n) for maintaining the topological ordering throughout all the edge additions. At the heart of the algorithm is a new approach for maintaining the ordering. Instead of attempting to place the nodes in an ordered list, they assign each node a label that is consistent with the ordering, and yet can be updated efficiently as edges are inserted. When the graph is dense, the algorithm is more efficient than existing

algorithms mentioned above. The idea used in their algorithm is quite different from previous algorithms. Typically, a topological ordering is maintained explicitly as either a linked list or an array. When adding an edge (u,v), the algorithm first checks whether u appears before or after v in the existing topological ordering; if u appears after v, then the array or linked list is updated so that u precedes v in the ordering, as is required by the insertion of edge (u,v). During the insertion, the algorithm modifies only vertices in the "affected region" of the list/array, i.e., those vertices that lie between *v* and *u*. The key to these algorithms is to efficiently discover which vertices in the affected region need to be moved.

This work is the only one we notice that the data structure can readily support edge deletions as well as edge insertions; however, performance guarantees apply only to executions consisting only of insertions.

5 Conclusions

If legacy codes cannot be sped up, multicore adoption can jeopardize decades of low-cost, continuous application performance improvements, a key enabler for substantial progress we have made in science and commerce. The goal of the seedling was to study the suitability of automatic parallelization techniques for parallelizing legacy codes. While we find that n-gram (i.e., repeated instruction sequences) parallelism does exist in the legacy binaries we studied, this inherent parallelism is not easily leveraged on mulitcore architectures. Such n-gram analysis can, however, assist compiler-based parallelism. Furthermore, we believe that focusing on the design of tools that examine input data patterns can provide valuable guidance in the extraction of high-order parallelism. Indeed, effective parallelism in code will vary for different inputs. Understanding how one many transform input data to maximize available parallelism is an ripe for deeper analysis.

6 Recommendations

We have identified a number of promising paths for extracting parallelism from legacy code by leveraging the relationship between compilers and microarchitectures. In particular, introducing accelerators for n-grams and hardware dependency analysis can help aid compiler-based parallelization. Furthermore, we believe that the nature of data consumed by legacy applications can provide strong hints as to the data dependencies that must be respected by efforts to extract task level parallelism.

We believe that answering the following two questions will aid in future attempts to automatically extract parallelism, and we believe that this approach is much more tenable than more invasive approaches that involve refactoring the legacy code in question. First, is it possible to derive or predict data dependencies at runtime based on an understanding of the high-level data organization of the application input (e.g., records, structures, objects, complex data structures, abstract data types)? Second, can we create "data compilers" that partition, transform, and convert data into portions that minimize data dependencies?

7 References

[AF07] D. Ajwani and T. Friedrich. Average-case analysis of online topological ordering. Proceedings of the 18th International Symposium on Algorithms and Computation (ISAAC). Lecture Notes in Computer Science, 4835:464–475, 2007.

[AFM08] D. Ajwani, T. Friedrich, and U. Meyer. An $O(n^{2.75})$ algorithm for incremental topological ordering. ACM Transactions on Algorithms (TALG), 4(4):Article No. 39, 2008.

[AHRSZ90] B. Alpean, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In Proceedings of the 1st Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 32–42, 1990.

[B90] F. Belik. An efficient deadlock avoidance technique. IEEE Transactions on Computers (TC), 39(7), 1990.

[BFG09] M. A. Bender, J. T. Fineman, and S. Gilbert. A new approach to incremental topological ordering. In Proceedings of the 19th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 1108–1115, 2009.

[BVZJA07] M. J. Bridges, N. Vachharajani, Y. Zhang, T. Jablin, and D. I. August. Revisiting the sequential programming model for multi-core. In Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 69-84, 2007.

[K04] I Katriel. On algorithms for online topological ordering and sorting. Technical Report MPI-I-2004-1-003, Max-Planck-Institut fur Informatik, 2004.

[KB06] I. Katriel and H. L. Bodlaender. Online topological ordering. ACM Transactions on Algorithms (TALG), 2(3):364–379, 2006.

[KM07] T. Kavitha and R. Mathew. Faster algorithms for online topological ordering. Technical report, CoRR, abs/0711.0251, 2007.

[KS74] D. E. Knuth and J. L. Szwarcfiter. A structured program to generate all topological sorting arrangements. Information Processing Letters (IPL), 2(6):153–157, 1974.

[LC07] H. F. Liu and K. M. Chao. A tight analysis of the katriel-bodlaender algorithm for online topological ordering. Theoretical Computer Science (TCS), 389(1-2):182–189, 2007.

[MNR93] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-line graph algorithms for incremental compilation. Proceedings of the International Workshop on Graph-Theoretic Concepts in Computer Science (WG), Lecture Notes in Computer Science, 790:70–86, 1993.

[MNR96] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. Information Processing Letters (IPL), 59(1):53–58, 1996.

[MV02] L. Michel and P. Van Hentenryck. A constraint-based architecture for local search. In Proceedings of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA), pages 83–100, 2002.

[OLB92] S. M. Omohundro, C. C. Lim, and J. Bilmes. The sather language compiler/debugger implementation. Technical Report TR-92-017, International Computer Science Institute, Berkeley, California, 1992.

[PK06] D. J. Pearce and P. H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. Journal of Experimental Algorithms (JEA), 11(7), 2006.

[PKH03] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In Proceedings of the 3rd International IEEE Workshop on Source Code Analysis and Manipulation (SCAM), 2003.

[RR94] G. Ramalingam and T. W. Reps. On competitive on-line algorithms for the dynamic priority ordering problem. Information Processing Letters (IPL), 51:155–161, 1994.

[RR96] G. Ramalingam and T. W. Reps. On the computational complexity of dynamic graph problems. Theoretical Computer Science (TCS), 158(1-2):233–277, 1996.

[T72] R. E. Tarjan. Depth-first search and linear graph algorithms. SIAM Journal of Computing (SICOMP), 1(2):146–160, 1972.

List of Acronyms, Abbreviations, and Symbols

Acronym	Description
ILP	Instruction Level Parallelism
DSWP	Decoupled Software Pipelining
PS-DSWP	Parallel Stage Decoupled Software Pipelining
LLVM	Low-Level Virtual Machine
DAG	Directed Acyclic Graph
SMP	Symmetric Multiprocessing
PDG	Program Dependence Graph
GIMP	GNU Image Manipulation Program
LLVM	Low-Level Virtual Machine
CP	Critical Path